

Model-Based Fault Localization in Large-Scale Computing Systems

Naoya Maruyama (*Tokyo Tech*)
Satoshi Matsuoka (*Tokyo Tech / National
Institute of Informatics*)

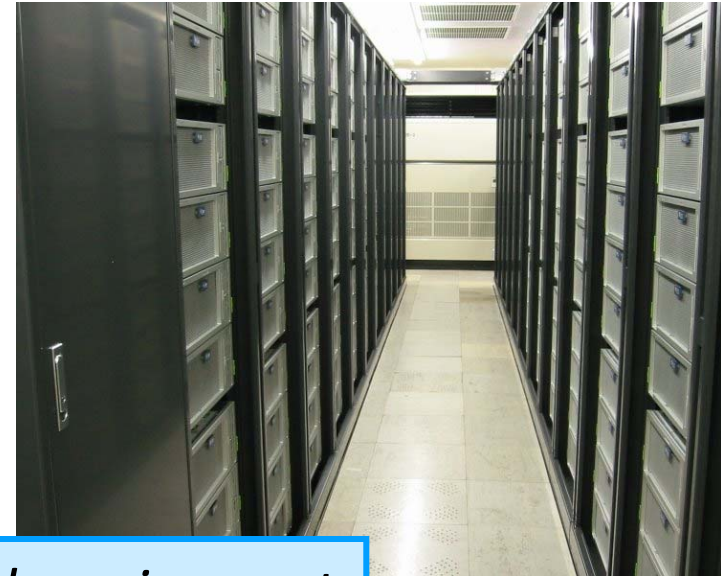
IPDPS'08 @ Miami

April 15, 2008

Background: Trends in HPC Systems

- *Scaling Up*

- Tokyo Tech TSUBAME supercomputer: >600 nodes, 10K cores
- IBM Blue Gene/L: 65K nodes
- InTrigger: 11 clusters distributed across Japanese universities



More failures, increased debugging costs

- *Diversity*

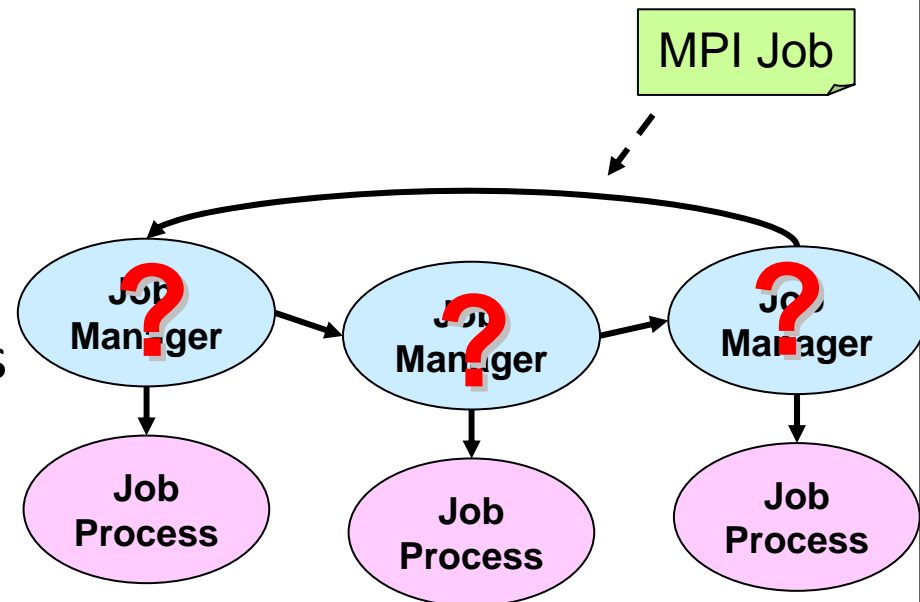
- Site-specific heterogeneous configurations of hardware and software components
 - e.g., MPI, Lustre, x86 CPUs, InfiniBand, SIMD Accelerators, GPUs.
- Low cost & flexible
- No dominant single vendor



Unexpected, hard-to-reproduce failures

Example: MPD on a Grid

- MPD: The default job manager for a popular MPI implementation (MPICH2)
 - Used on a tightly coupled parallel system → No problem
 - On geographically distributed machines → Jobs do not start but hang with a high probability
- Cause: A very simple bug and naïve error-recovery code
 - Bug: Missing check of recv return values
 - No error output, but only terminated the connections



Existing Approaches

- The “printf” debugging
 - Inserts printf → reruns the system → manual checking of output → re-inserts printf → ...
 - Too ad-hoc for current and future large-scale supercomputers and Grids
- Parallel debuggers
 - e.g., TotalView
 - Difficult to reproduce nondeterministic failures

Objective

Assisting fault localization by finding anomalous behaviors

- Automate as much as possible
- Fine-grained localization
- Scalability with a large number of processes
- Minimize assumptions on target systems

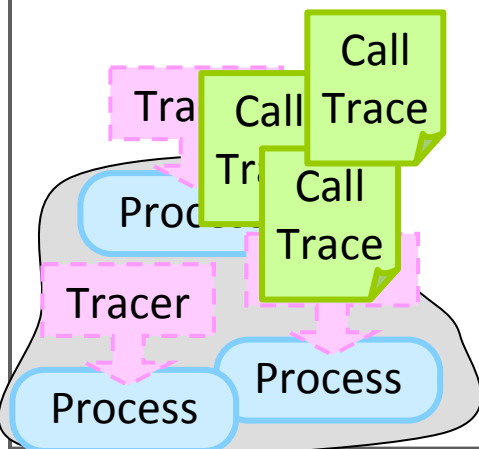
Our Approach

Assumes *historical similarities* in execution behavior

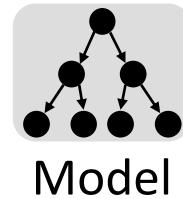
- “Processes should exhibit repetitive, self-similar behavior”
- Esp. true in daemon-like processes

1. Data collection
2. Behavior modeling
3. Anomaly detection

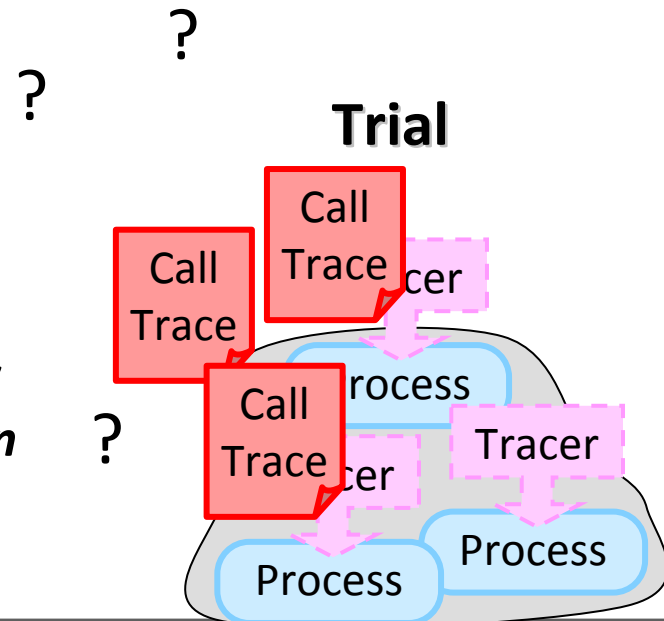
Normal



Behavior
Modeling



Anomaly
Detection



Contributions

- Model-based fault localization for large-scale computing systems
 - Trace collection → Behavior modeling
 - Finds suspicious function calls by detecting anomalies in function traces
- Demonstrated its effectiveness with the MPD hang-up bug
 - Detected an erroneous network closing among 78 processes, spanning 3 sites

Outline

- ✓ 1. Introduction
2. Data collection
3. Modeling
4. Fault localization
5. Case study
6. Conclusion

Data Collection

```
...  
ENTER func_addr 0x819967c timestamp 12131002746163258  
LEAVE func_addr 0x819967c timestamp 12131002746163936  
ENTER func_addr 0x819967c timestamp 12131002746164571  
LEAVE func_addr 0x819967c timestamp 12131002746165197  
ENTER func_addr 0x819967c timestamp 12131002746165828  
LEAVE func_addr 0x819967c timestamp 12131002746166395  
LEAVE func_addr 0x80de590 timestamp 12131002746166938  
ENTER func_addr 0x819967c timestamp 12131002746167573  
...
```

- Appends an entry at each call and return
 - addresses, timestamps
- Allows function-level analysis
 - *e.g.*, “Function X is likely anomalous.”
- Allows context-sensitive analysis
 - *e.g.*, “Function X is anomalous only when called from function Y.”

Outline

- ✓ 1. Introduction
- ✓ 2. Data collection
3. Modeling
4. Fault localization
5. Case study
6. Conclusion

Modeling for Fault Localization

Observation

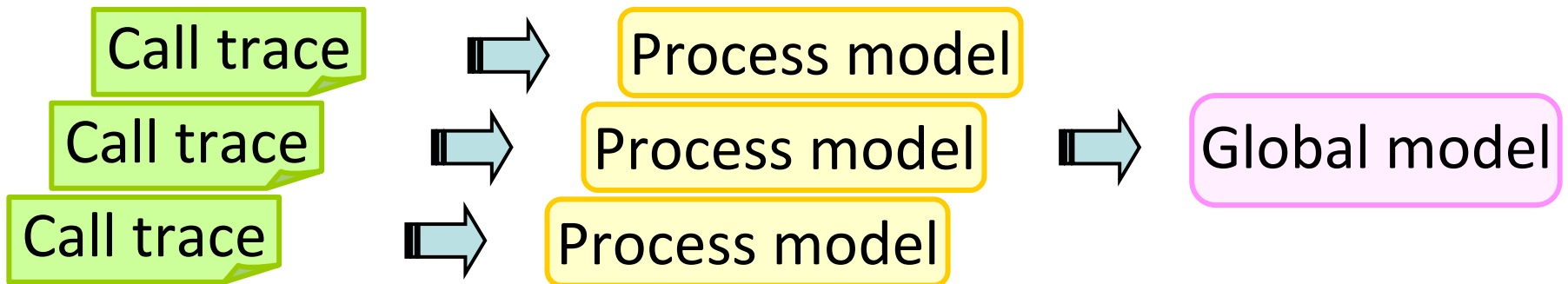
- Usually called, but not when a failure occurred → *Suspicious*
 - e.g., logic bugs (the failure at the Tokyo Stock Exchange on 2/9/2008)
- Rarely called, but was actually called → *Suspicious*
 - e.g., little exercised code



Modeling by function call probabilities

Modeling Steps

1. Obtain **normal** per-process traces
2. Derive *process models*
3. Derive *global models*

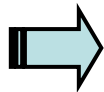


Deriving Process Models

- **Input:** per-process trace obtained from a single normal execution
- **Output:** a call tree annotated with estimated call probabilities

Per-Process Trace

```
...  
CALL    F  
CALL    G  
RETURN  G  
CALL    H  
RETURN  H  
RETURN  F  
...
```

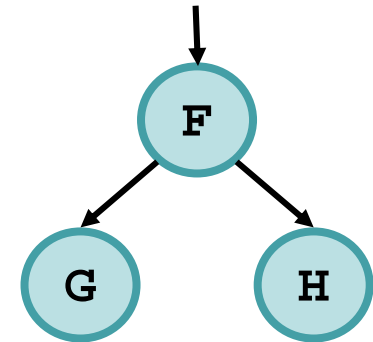


Call Counts

	Count
F	2
G	2
H	1



Annotated Call Tree



$$P(G) = 1.0 \quad P(H) = 0.5$$

Model Refinement

- **Idea:** Allocate different models to different phases of execution
- Typical phases in distributed processes
 - Initialization
 - Finalization
 - Event handler executions

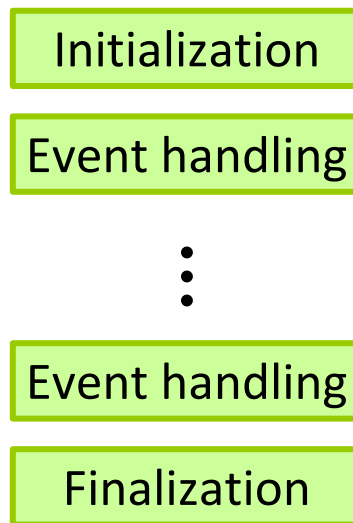
Per-process trace

```
...  
CALL    F  
CALL    G  
RETURN  G  
CALL    H  
RETURN  H  
RETURN  F  
...
```

Decompose



Execution units



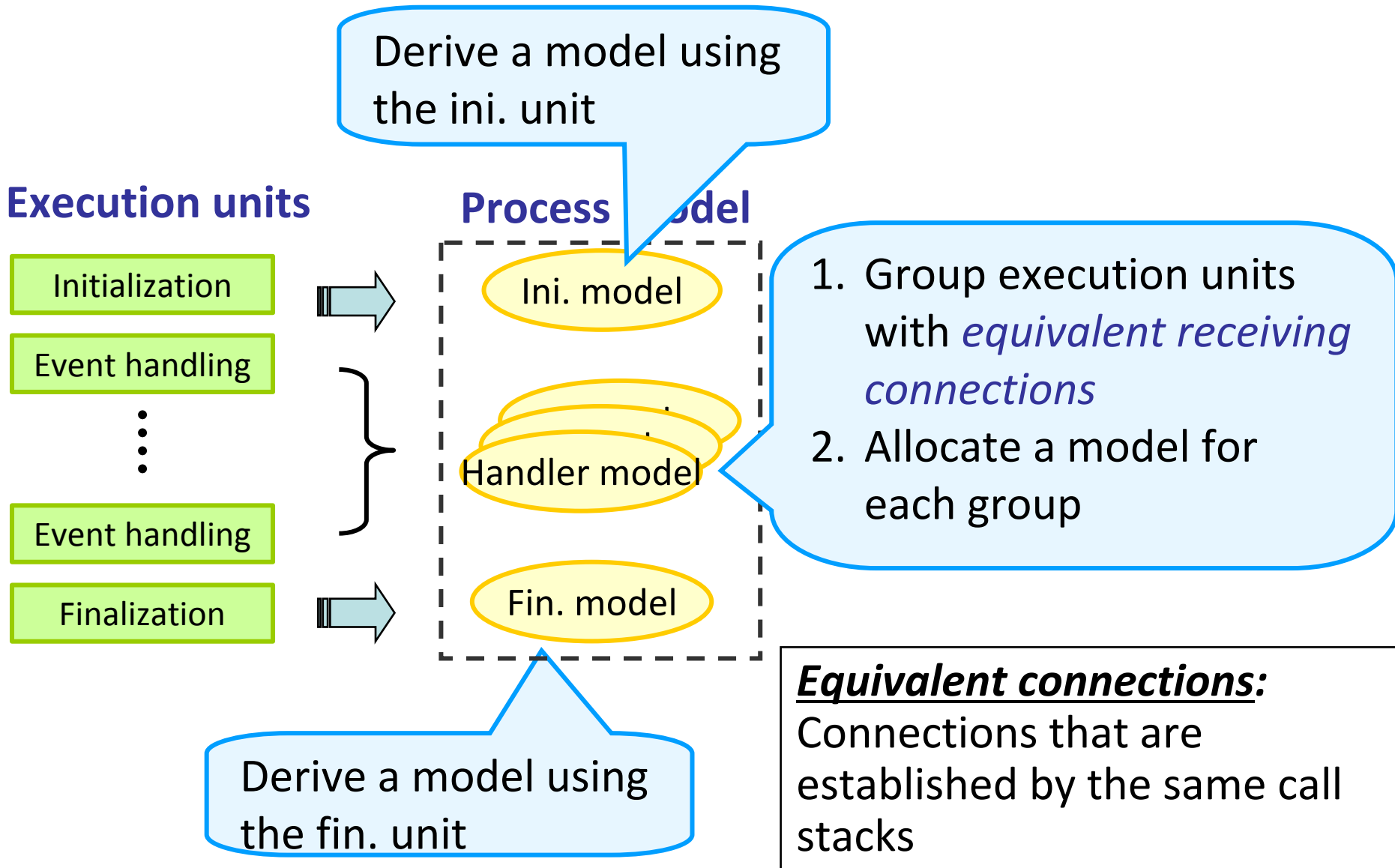
Decomposition Steps

Events:

Arrivals of remote messages

1. Identify the main loop
 - Detect the outermost loop by finding repeatedly appearing same stacks
2. Decomposes traces before calls to `recv` in the main loop

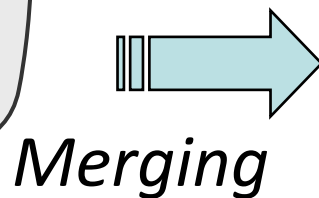
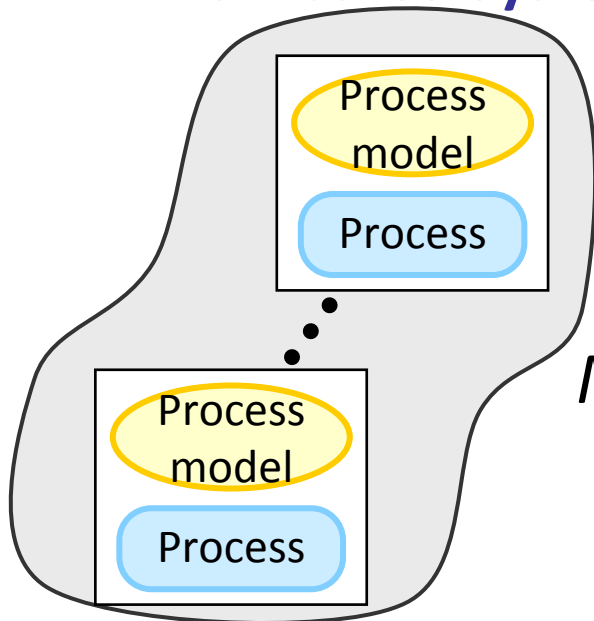
Phase-Specific Model Allocation



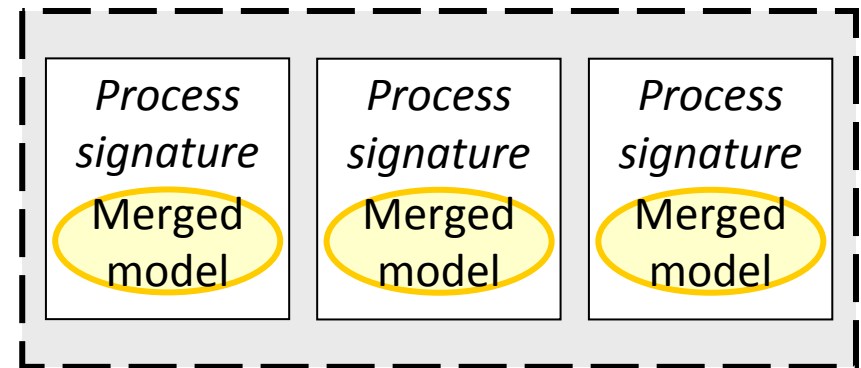
Deriving Global Models

- **Input:** process models for all processes
- **Output:** a compound model effectively including all the process models (i.e., a *global model*)
 - Too much to retain all the individual models
 - Merge models with the same *process signature* by accumulating call counts

Distributed System



Global Model



Process Signature:

Stacks that establish initial connections

Outline

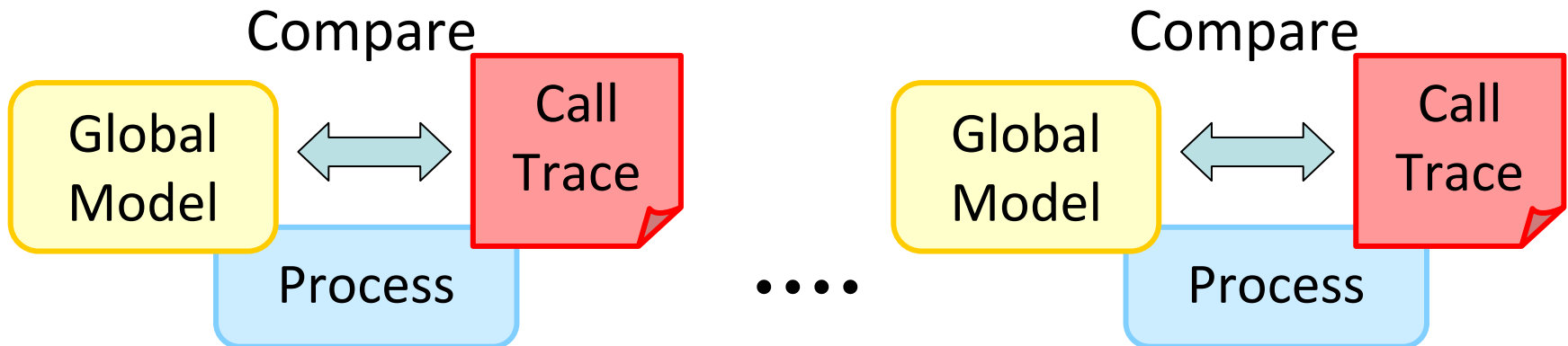
- ✓ 1. Introduction
- ✓ 2. Data collection
- ✓ 3. Modeling
4. Fault localization
5. Case study
6. Conclusion

Localizing Faults Using the Derived Model

0. Deploy the pre-derived global model

System Failure

1. Decomposes failure traces into execution units
2. Quantifies how different each unit is from the model as a *suspect score*
3. Rank all units with their suspect scores



Suspect Scores

- Quantitative indicators of suspiciousness of execution units
 - High-probability functions → high scores if not called
 - Low-probability functions → high scores if called

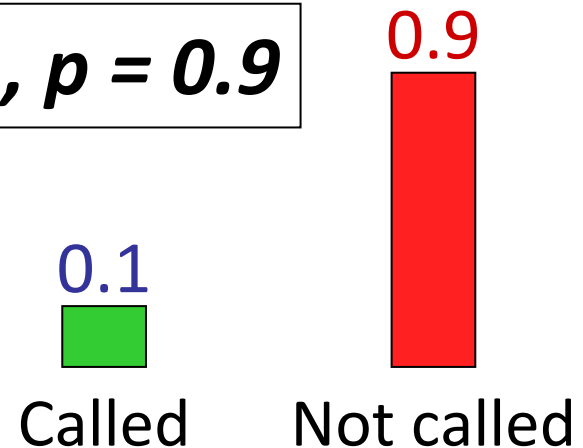
1. For each call, F , compute its score S_F

- Called → $S_F = 1 - P_F$
- Not called → $S_F = P_F$

E.g., $p = 0.9$

2. Total score of a unit

- $S = \text{mean}(S_F)$



Outline

- ✓ 1. Introduction
- ✓ 2. Data collection
- ✓ 3. Modeling
- ✓ 4. Fault localization
5. Case study
6. Conclusion

Case Study

- *Problem:* The MPD bug
 - Cause: Missing return value check of a recv syscall
 - Naïve error handling code only closed the network connection when a partial message is received, but not terminated the system nor notified the user
 - Occurs with version 1.0.5p4, fixed with later versions
 - Discovered through manual printf debugging by Hideo Saito
- *Purpose:* Demonstrate how effective our technique is for discovering these misbehaviors

Trace Collection and Modeling

- Function tracer

- Library interpositioning
- Compiler- and runtime-provided function hooks

- Normal traces

- Traced the system while running on a single cluster



Global model

- Failure traces

- Traced the system while running on a grid consisting of 78 nodes, spanning 3 sites



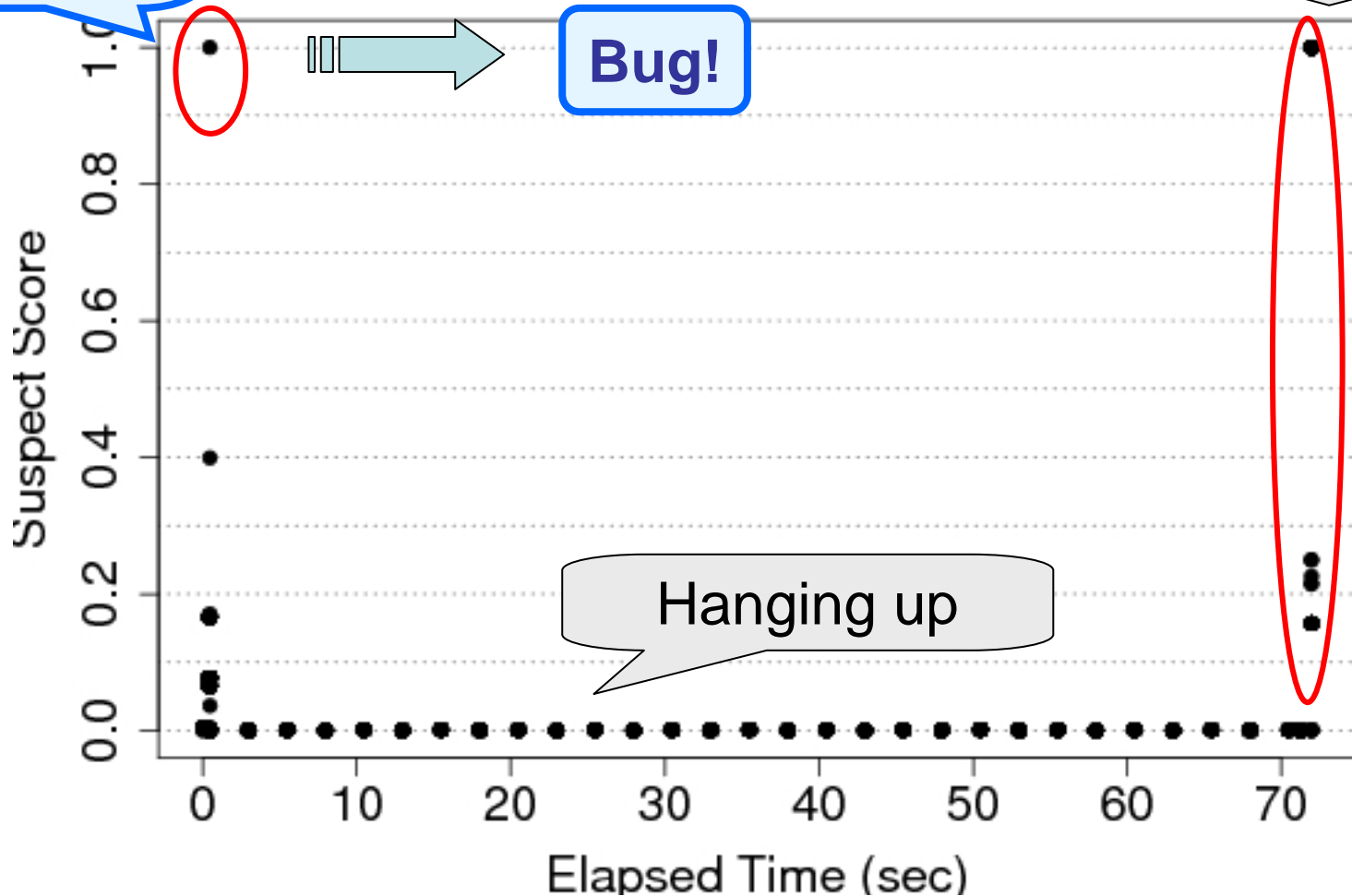
Suspect scores

Localization Results

Connection closed

Further manual examination

Due to abrupt system termination



Bug!

Hanging up

Outline

- ✓ 1. Introduction
- ✓ 2. Data collection
- ✓ 3. Modeling
- ✓ 4. Fault localization
- ✓ 5. Case study
- 6. Conclusion

Related Work

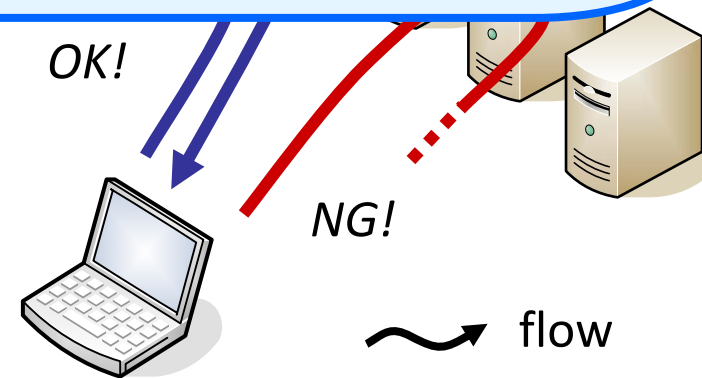
Fault localization via anomaly detection in distributed flows

- [Chen04], [Kiciman05], [Barham04], [Cohen04], [Yuan06], [Renieris03]
- Flows: HTTP requests, RPCs

1. Discover **distributed flows** among components by matching distributed traces

2. Find suspicious components by comparing **normal flows** and **failure flows**

Q. *Scalability?* Can handle tens of thousands of distributed machines?



Our approach:

Only use local observations



(Mostly-) **Distributed** modeling and fault localization

Summary

- Fault localization by modeling system behaviors and detecting anomalies
 - Detecting anomalous traces by comparing failure traces with the model
- Can detect unusually (not-)called functions
 - e.g., logic bugs, less exercised functions
- Preliminary evaluation with a case study in a Grid environment
 - Automatically detected an erroneous network closing among 78 nodes